CSN08x14

Scripting for Cybersecurity and Networks

Lecture 3:

Python dictionaries; tuples; reading from files; hashing

Today's Topics

- Dictionaries
- Tuples
- Reading from external files
- Crypto Hashes – Hashing in Python
- Hash Password Recovery script – dict_crack.py

Basic Data Types/Object Types:

- **Numbers:** `bool, int, float`
- **String:** `str`

Last 2 Weeks

- **Collection Objects:**
  - ***List:*** `list`
  - ***Dictionary:*** `dict`
- **Tuple:** `tuple`

This Week

Scripting for Cybersec & Networks

Go to www.menti.com

code **15 22 57**

The problem with lists…

a problem with lists

- Accessing Lists by index – can't tell what you are indexing

    - bond = [c', '*classified*', 37]

Person Object/Record:
Name, DoB, Age

```
>>> print (bond[0], bond[2])
        Bond. James Bond 37
```

Person's Age
Accessed via
index 2

- Issues?
    - *Can't tell what you are indexing – Name = index/offset 0*
    - *No association between index and list item value*
    - *Not intuitive for associative collection*
    - *Portability: different *people* may have *age* recorded at a different offset*

a problem with lists (ctd)

- Could use nested lists with "keys":

```
>>> bond = [['name', 'Bond. James Bond'],
            ['dob', '*classified*'],
            ['age', 37] ]


>>> print (bond[0][1], bond[2][1])

     Bond. James Bond 37

>>> for (key, value) in bond:

       if key == 'age': print(value)

     37
```

Person's Age
Accessed via match on key string 'age'

a problem with lists (ctd)

- Nested lists with "keys":

```
bond = [['name', 'Bond. James Bond'],
        ['dob', '*classified*'],
        ['age', 37] ]
```

- Good
  - *Pretty good solution – associates a key with a value*
  - *Lookup the key and get the value back – could implement in a function*
  - *Quite scalable – can add key:value pairs dynamically*

- Issues
  - *Verbose and clumsy*
  - *Performance – have to access each item in list – go through entire list for last item*

a problem with lists (ctd)

- Nested lists with "keys":

```
bond = [['name', 'Bond. James Bond'],
        ['dob', '*classified*'],
        ['age', 37] ]
```

Good

- Pretty good solution – associates a key with a value
- Lookup the key and get the value back – could implement in a function
- Quite scalable – can add key:value pairs dynamically

Issues

- Verbose and clumsy
- Performance – have to access each item in list – go through entire list for last item

Python dictionaries

Python Dictionary

- Better way to associate keys and values in a data structure: the built in Python data type – **Dictionary (dict)**.
  - *Can use different data types as index/key*

```
>>> bond = {'name': 'Bond. James Bond',
            'dob': '*classified*',
            'age': 37 }
```

Keys
Basic Python Objects - usually strings

Values
Any Object type

dicts were "unordered" collections, meaning the keys could be returned in any order. Since Python 3.7, dicts are ordered, insertion order is guaranteed to be maintained.

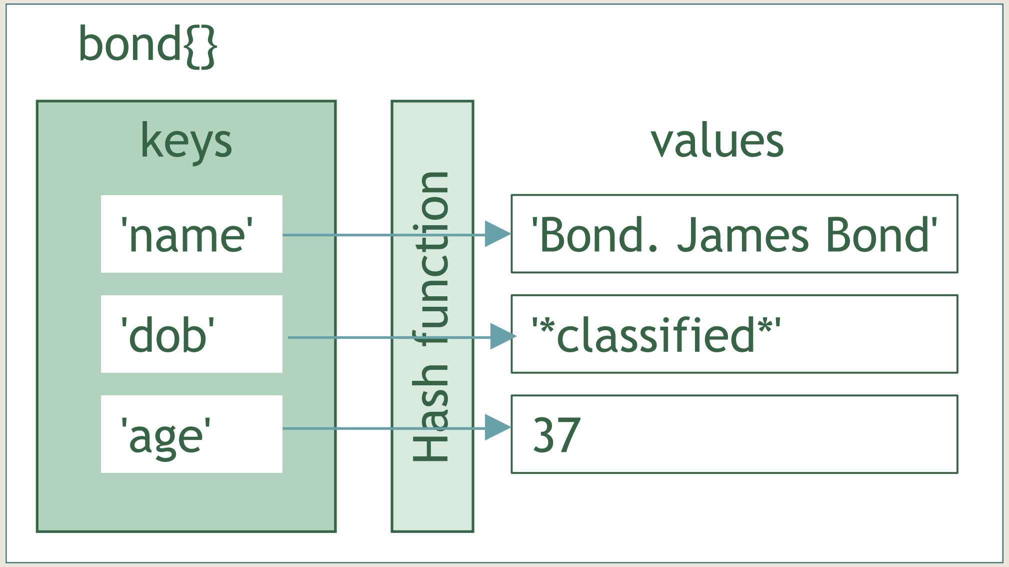
Python Dictionary

- Python's implementation of "hash tables"
- Keys must be unique within a dictionary
- Efficient performance – Direct access to Values.

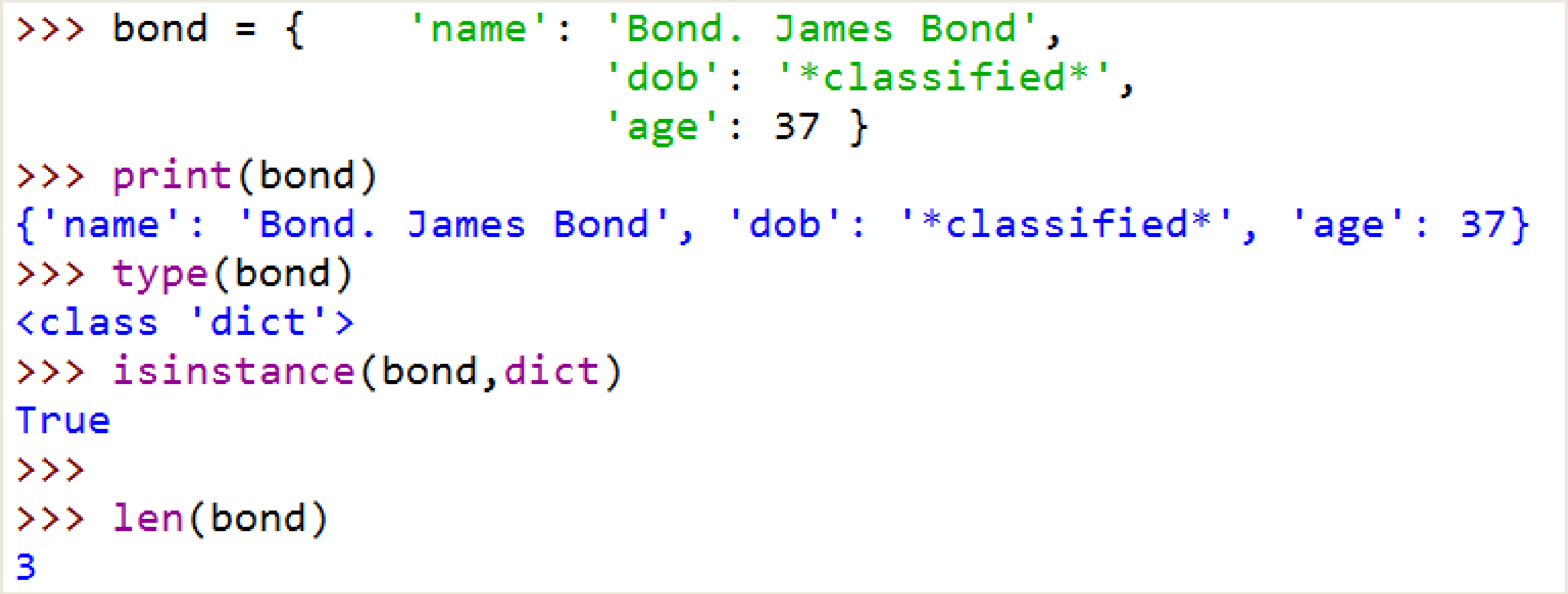
```
>>> bond = {'name': 'Bond. James Bond',
            'dob': '*classified*',
            'age': 37 }
```

```
>>> bond['name']
   Bond. James Bond
>>> bond['age']
   37
```

Index by Key
Returns Value



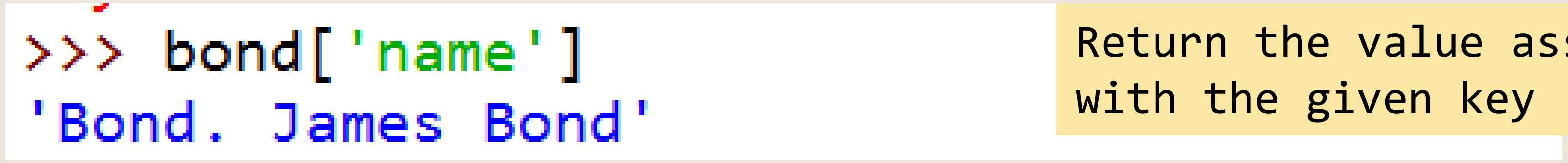
Remember: Python uses same syntax for different data types!



```
>>> bond = {    'name': 'Bond. James Bond',
                                'dob': '*classified*',
                                'age': 37 }
>>> print(bond)
{'name': 'Bond. James Bond', 'dob': '*classified*', 'age': 37}
>>> type(bond)
<class 'dict'>
>>> isinstance(bond,dict)
True
>>>
>>> len(bond)
3
```

Accessing Dictionary Objects

- Same as [] operator for Lists & Strings but use the **key** instead of index value



```
>>> bond['name']
'Bond. James Bond'
```

```
Return the value associated
with the given key
```

Checking for items in Dictionary

- **in** Membership operator – checks if key in dict object (**not in** checks absence)

```
>>> bond = {'name': 'Bond. James Bond', 'dob': '*classified*', 'age': 37 }
>>> 'name' in bond
True
>>> 'weight' in bond
False
```

- could use in / not in with if statements to avoid errors

```
>>> if 'age' in bond:
        print (f'Age -> {bond["age"]}')
Age -> 37
>>> if 'weight' not in bond:
        print ('Key not found')
Key not found
```

...but this isn't very "pythonic"...

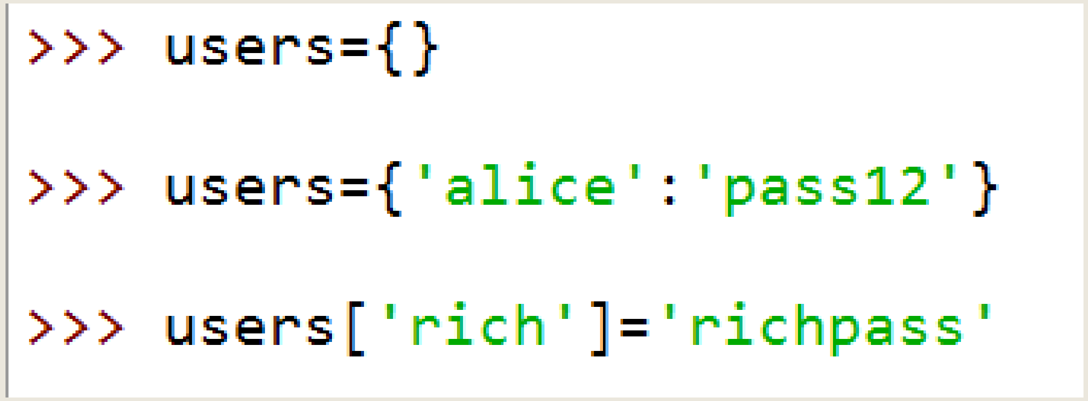
.get() method: Checking for/getting items in Dictionary

`dict.get(<key>[,def])`

- *Takes key plus optional 2nd argument*
- *Returns value of key, or 2nd argument if not found*

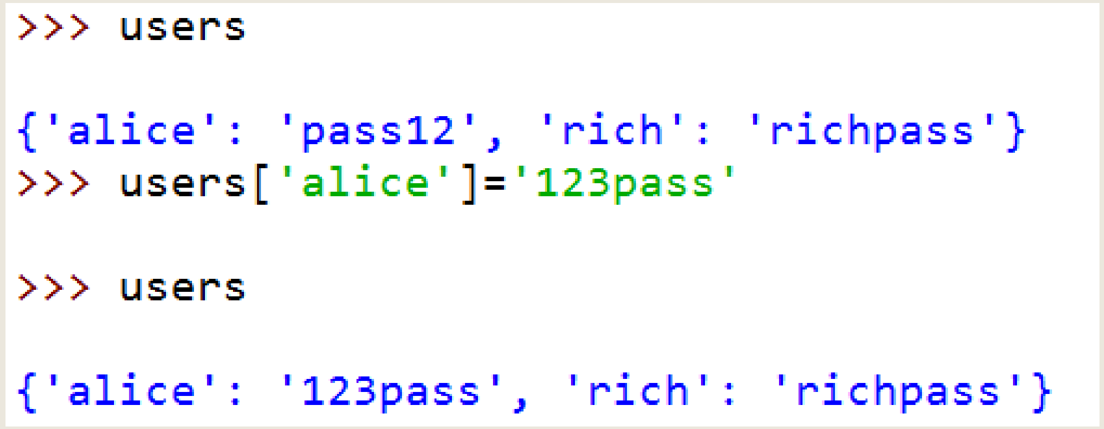
```
>>> bond.get('age', 'Age not found')
37
>>> bond.get('weight')
>>> bond.get('weight', 'Weight not found')
'Weight not found'
```

Creating a Dictionary, adding and changing values



```
>>> users={}

>>> users={'alice':'pass12'}

>>> users['rich']='richpass'
```

```
# Create Empty Dictionary

# Create with a key:value pair

# add another entry
```

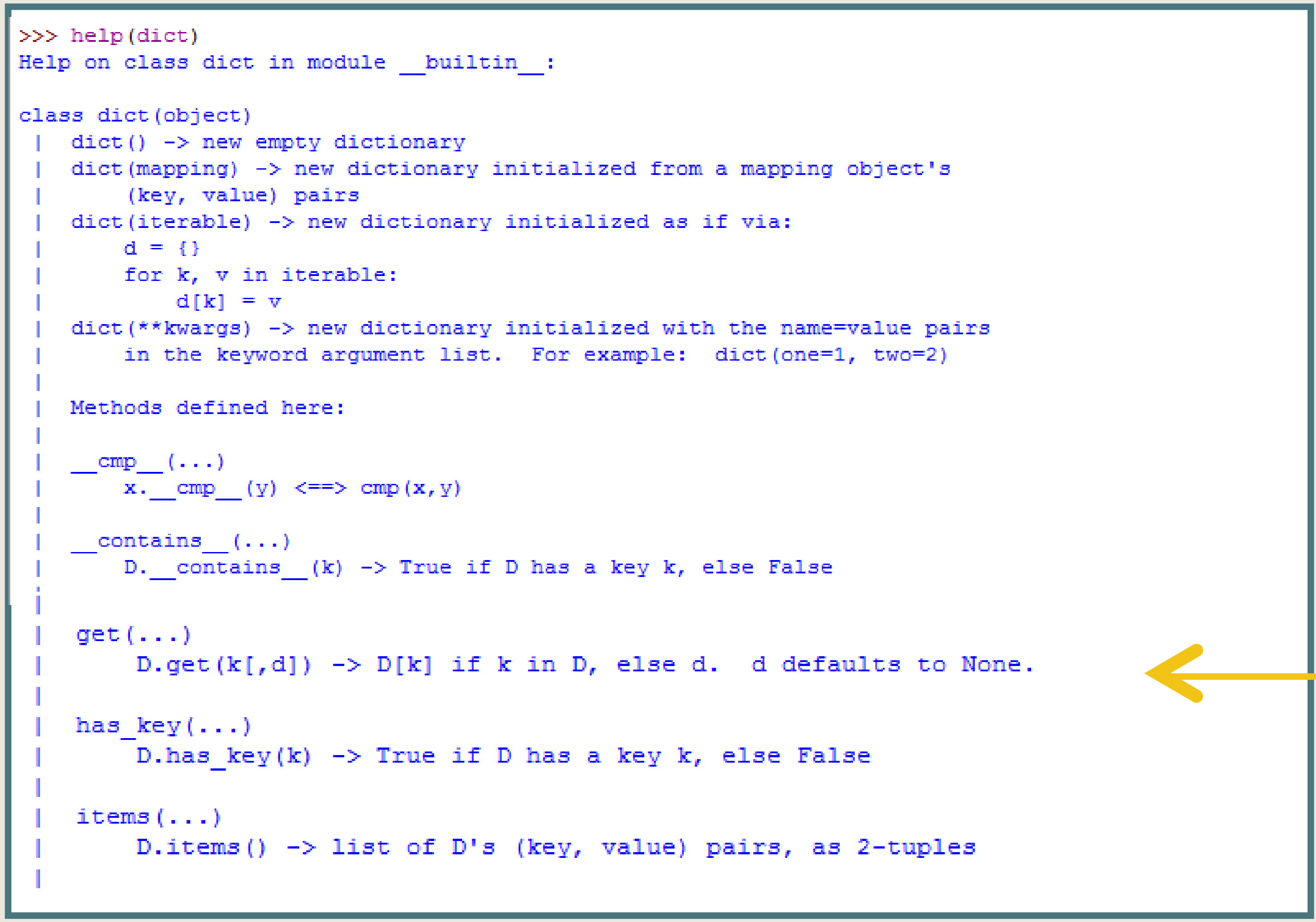


```
>>> users

{'alice': 'pass12', 'rich': 'richpass'}
>>> users['alice']='123pass'

>>> users

{'alice': '123pass', 'rich': 'richpass'}
```

```
# changing value for 'alice'
```

Dictionary Object specific Methods

- Use BIFs help(), dir() to list methods:

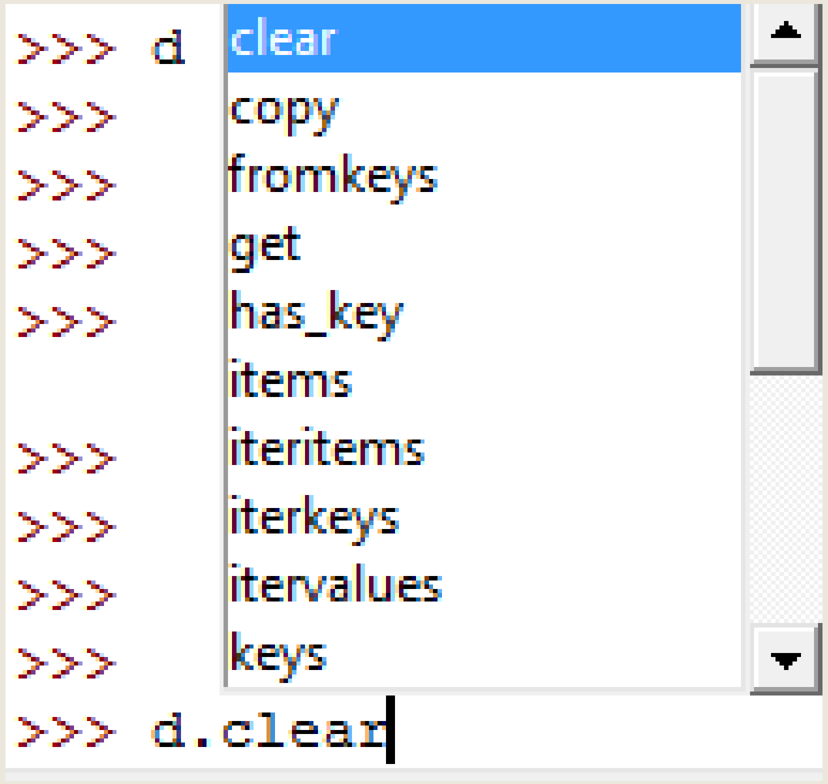


```
>>> help(dict)
Help on class dict in module __builtin__:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Methods defined here:
 |
 |  __cmp__(...)
 |      x.__cmp__(y) <==> cmp(x,y)
 |
 |  __contains__(...)
 |      D.__contains__(k) -> True if D has a key k, else False
 |
 |  get(...)
 |      D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.
 |
 |  has_key(...)
 |      D.has_key(k) -> True if D has a key k, else False
 |
 |  items(...)
 |      D.items() -> list of D's (key, value) pairs, as 2-tuples
 |
```

dict Specific
Named methods

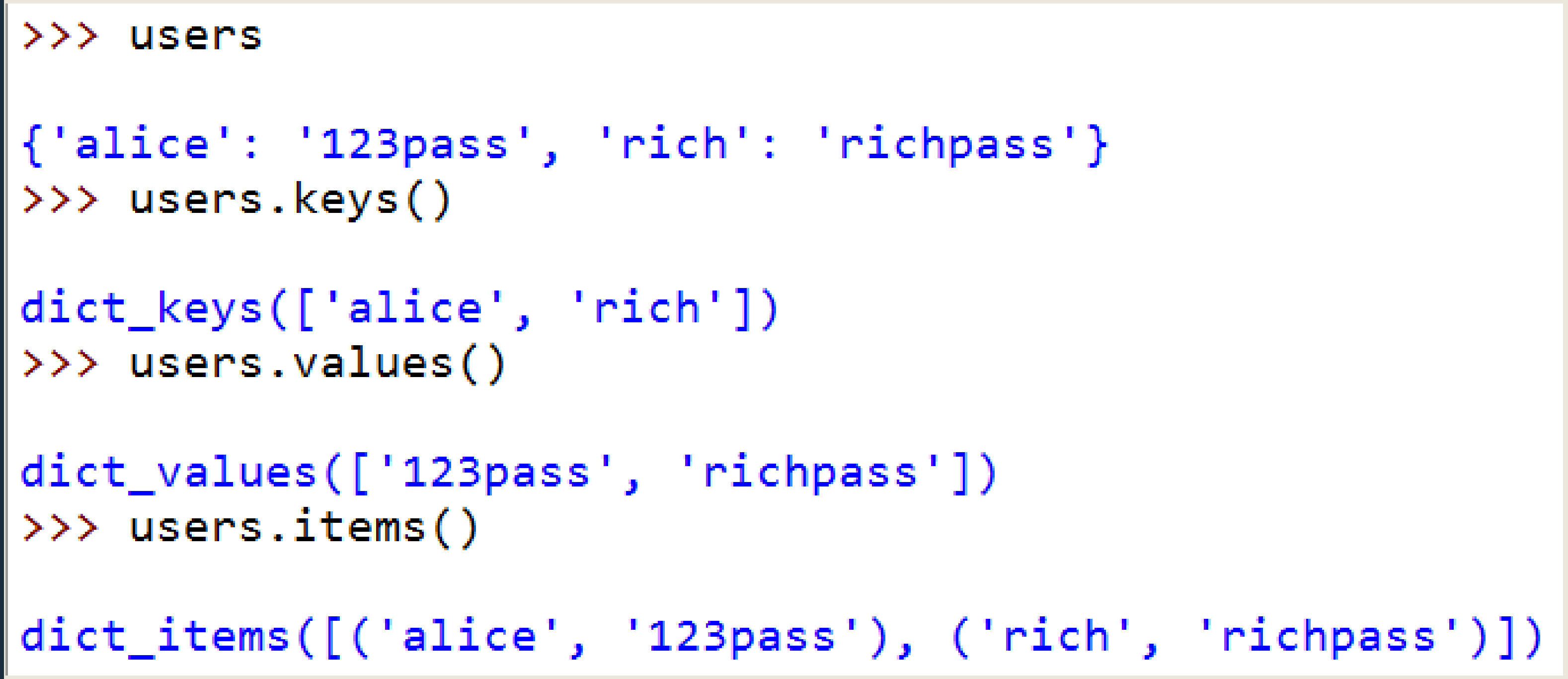
Dictionary Object Specific Methods

- Methods of the dict object can manipulate keys/values
- Dictionary Methods
  dict.<CTRL+SPACE>



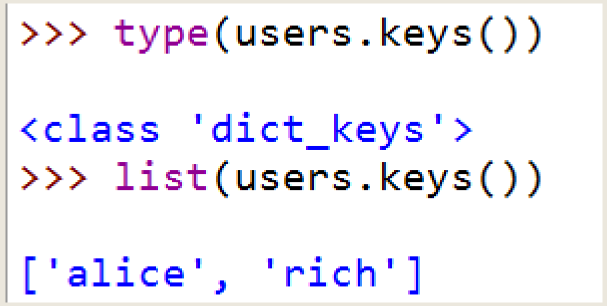
```
>>> d    clear
>>>      copy
>>>      fromkeys
>>>      get
>>>      has_key
         items
>>>      iteritems
>>>      iterkeys
>>>      itervalues
>>>      keys
>>> d.clear
```

.keys(), .values(),.items() methods



```
>>> users

{'alice': '123pass', 'rich': 'richpass'}
>>> users.keys()

dict_keys(['alice', 'rich'])
>>> users.values()

dict_values(['123pass', 'richpass'])
>>> users.items()

dict_items([('alice', '123pass'), ('rich', 'richpass')])
```

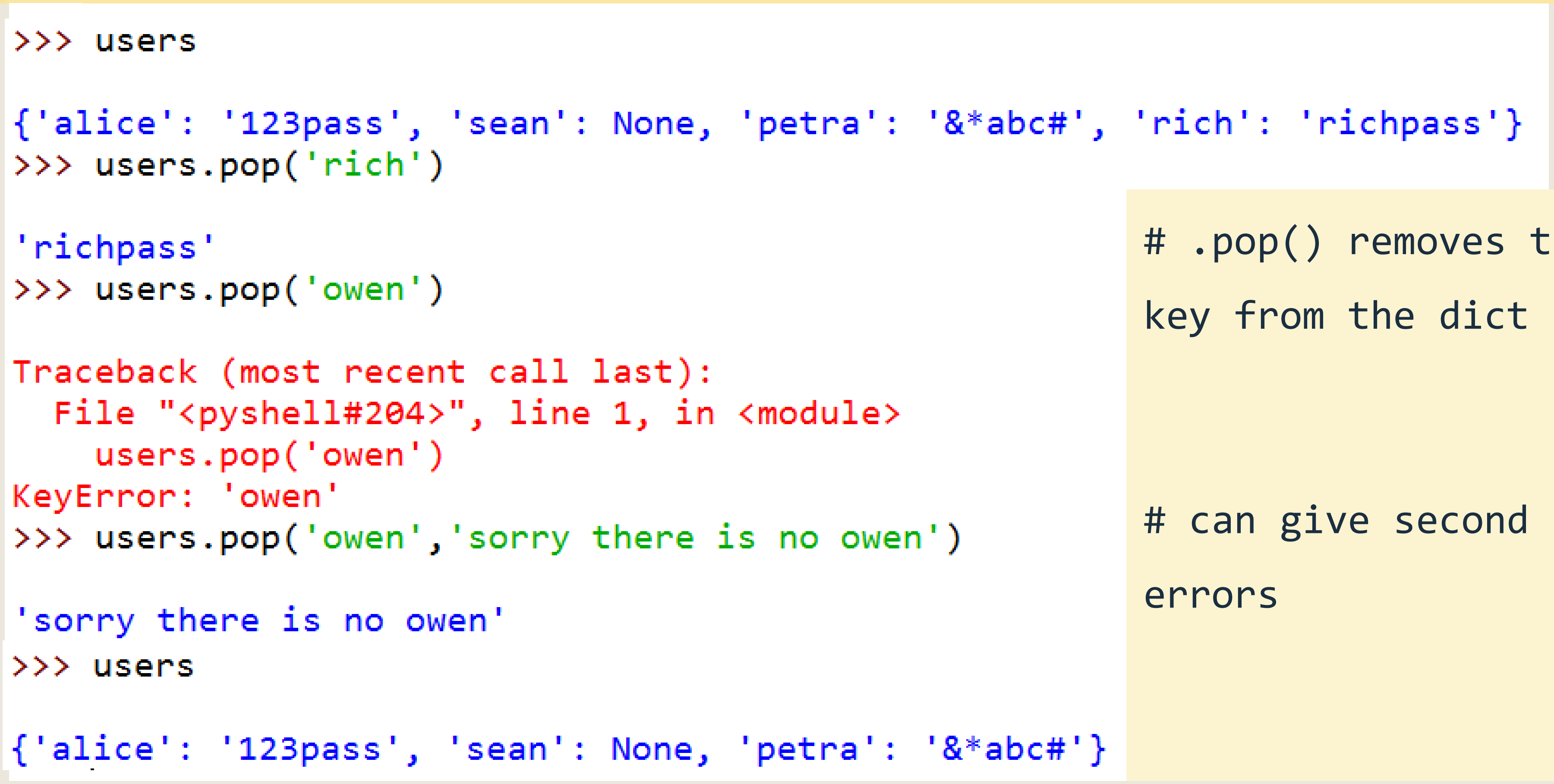
```
# .keys() returns all keys

# .values() returns all values

# .items() returns key:value
pairs as tuples
```



```
>>> type(users.keys())

<class 'dict_keys'>
>>> list(users.keys())

['alice', 'rich']
```

```
# use list() function to convert from specific
# object type to list
```

pop(): Removing from dicts



```
>>> users

{'alice': '123pass', 'sean': None, 'petra': '&*abc#', 'rich': 'richpass'}
>>> users.pop('rich')

'richpass'
>>> users.pop('owen')

Traceback (most recent call last):
  File "<pyshell#204>", line 1, in <module>
    users.pop('owen')
KeyError: 'owen'
>>> users.pop('owen','sorry there is no owen')

'sorry there is no owen'
>>> users

{'alice': '123pass', 'sean': None, 'petra': '&*abc#'}
```

```
# .pop() removes the value of the given
key from the dict and returns it

# can give second argument to trap
errors
```

Q: Where have we seen this behaviour before?

.copy(): Copying dict objects

```
>>> rob = bond.copy()

>>> rob
{'name': 'Bond. James Bond', 'dob': '*classified*', 'age': 37}
```

- Shallow copy – if there are sub-collections, they are referenced, not copied
- Use copy.deepcopy(dict) to copy subcollections

Could use
rob = bond
to copy but as with lists this would be different pointer to same stored dict,
not a different object

Dictionary Iteration

```
>>> bond = {'name': 'Bond. James Bond',
         'dob':'*classified*', 'age':37}
```

Used to Operate on each object in a dictionary

- `for in` built in construct

```
for key in dictionary:
        do something with  the key
```

```
>>> for k in bond:
        print(f'{k} -> {bond[k]}')
```

b is assigned to each key in Dictionary in turn

Indexing gets each value by key

```
name -> Bond. James Bond
dob -> *classified*
age -> 37
```

Since Python 3.7, insertion order is maintained. In earlier versions, dictionaries were unordered collection objects

More Efficient Iteration for dictionaries

- **For... in...** (used with items(), values() or keys())

```
for key, value in dictionary.items():
      do something with  the key/value
```

values(), keys(), items() methods can be used like this

```
>>> for k,val in bond.items():
    print(f'{k} -> {val}')
```

key/value automatically assigned to each key and value pair in Dictionary

No need to access Dictionary again

Dictionary Comprehension

- Similar to list comprehension
- Syntax:

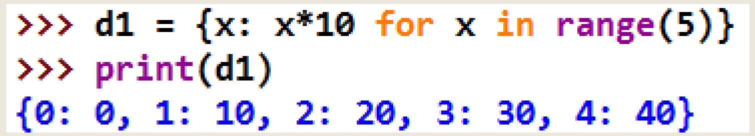
```
d = {key: value for key in iterable}
```

Enclosed in curly braces

Two expressions separated by a colon

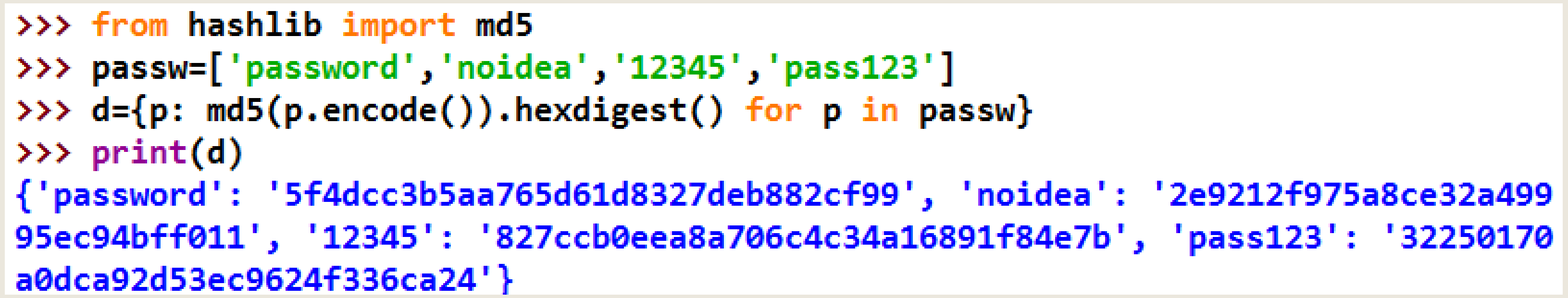
The iterable is usually a list or a range (not a dict)

- Example:



```
>>> d1 = {x: x*10 for x in range(5)}
>>> print(d1)
{0: 0, 1: 10, 2: 20, 3: 30, 4: 40}
```

- Complex example:

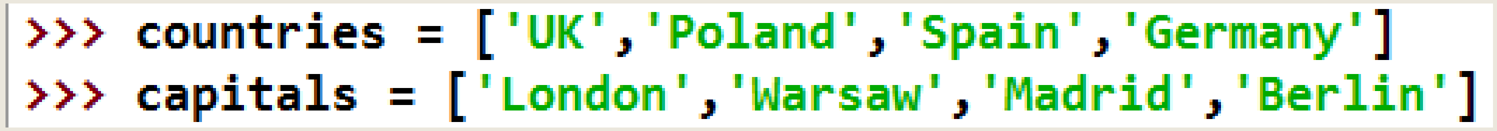


```
>>> from hashlib import md5
>>> passw=['password','noidea','12345','pass123']
>>> d={p: md5(p.encode()).hexdigest() for p in passw}
>>> print(d)
{'password': '5f4dcc3b5aa765d61d8327deb882cf99', 'noidea': '2e9212f975a8ce32a499
95ec94bff011', '12345': '827ccb0eea8a706c4c34a16891f84e7b', 'pass123': '32250170
a0dca92d53ec9624f336ca24'}
```

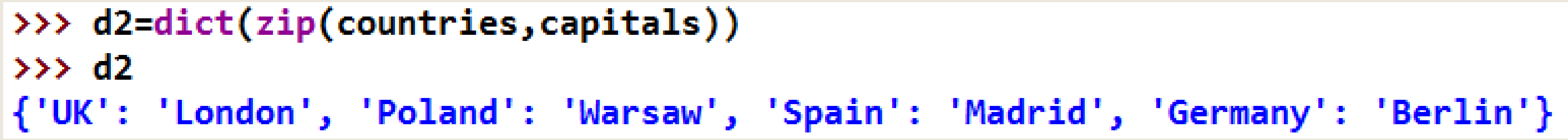
- See https://www.smallsurething.com/list-dict-and-set-comprehensions-by-example/, http://www.diveintopython3.net/comprehensions.html

zip(): Creating a dictionary from two lists

- With the zip() function we can merge two lists into a dictionary
- Example:

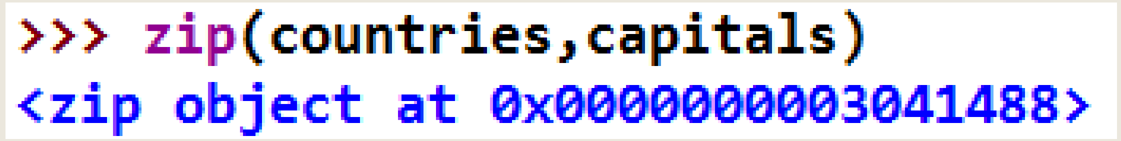


```
>>> countries = ['UK','Poland','Spain','Germany']
>>> capitals = ['London','Warsaw','Madrid','Berlin']
```



```
>>> d2=dict(zip(countries,capitals))
>>> d2
{'UK': 'London', 'Poland': 'Warsaw', 'Spain': 'Madrid', 'Germany': 'Berlin'}
```

- Explanation:
  - *zip(a,b) creates a zip object*



```
>>> zip(countries,capitals)
<zip object at 0x0000000003041488>
```

  - *dict() converts the object into a dictionary*

- See http://www.bogotobogo.com/python/python_dictionary_comprehension_with_zip_from_list.php

Sorting on Keys: sorted()

```
>>> bond = {'name': 'Bond. James Bond',
          'dob':'*classified*', 'age':37}
```

- **sorted() BIF**

```
for key, value in sorted(dictionary.items()):
       do something with  ordered key/value
```

```
>>> for k,val in sorted(bond.items()):
print(f'{k} -> {val}')
```

k/val sorted, then automatically assigned to each key and value pair in Dictionary

```
age -> 37
dob -> *classified*
name -> Bond. James Bond
```

Ordered (alphabetically) by Key

Usage of Dictionaries

- Typically used for large amounts of data
- Where we want to associate data with a key value
- Performance advantage for large data sets
- Sequence operators won't work – no slice [:]
  (because dicts are unordered!)

Usage of Dictionaries ctd

- Especially good where we have a lot of similarly structured data

  e.g.

  - IP Addresses -> traffic/server requests from parsing a web server log
  - Hash password lookup table
  - lots of passwords and their hashes
  - lots of countries and their capitals

  - Think carefully what should be used as key and what as value!

Tuples

Python Tuple

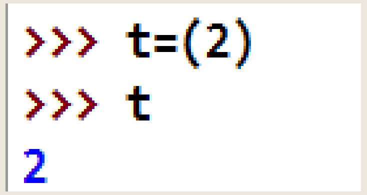
- Fixed size sequence object – like a database record
- Immutable – cannot be changed

```
 >>> t = ()                          # empty tuple
 >>> t = (2, 'curry', 333)           # new tuple
 >>> print (t)                       # quick print
 (2, 'curry', 333)
>>> print (t[1])                     # access by offset
'curry'
```

Tuples use round brackets

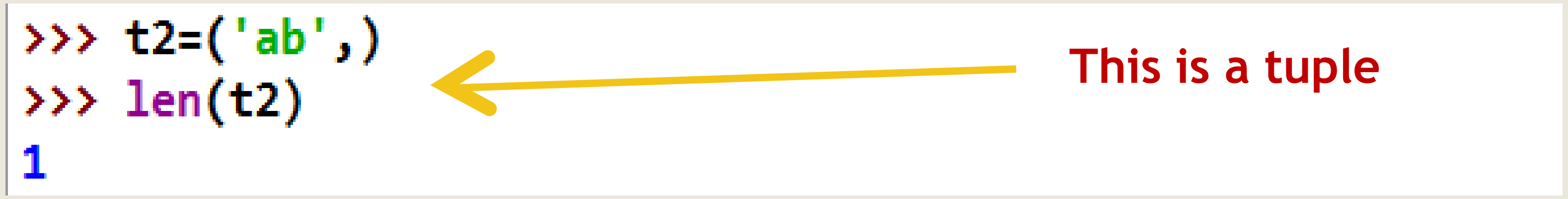
One-element tuples

- a tuple with one element must be defined with a comma



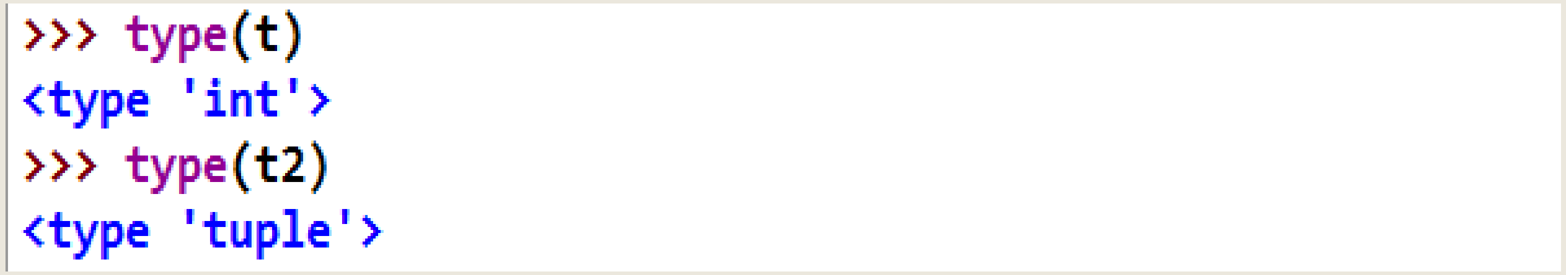
```
>>> t=(2)
>>> t
2
```

This is not a tuple



```
>>> t2=('ab',)
>>> len(t2)
1
```

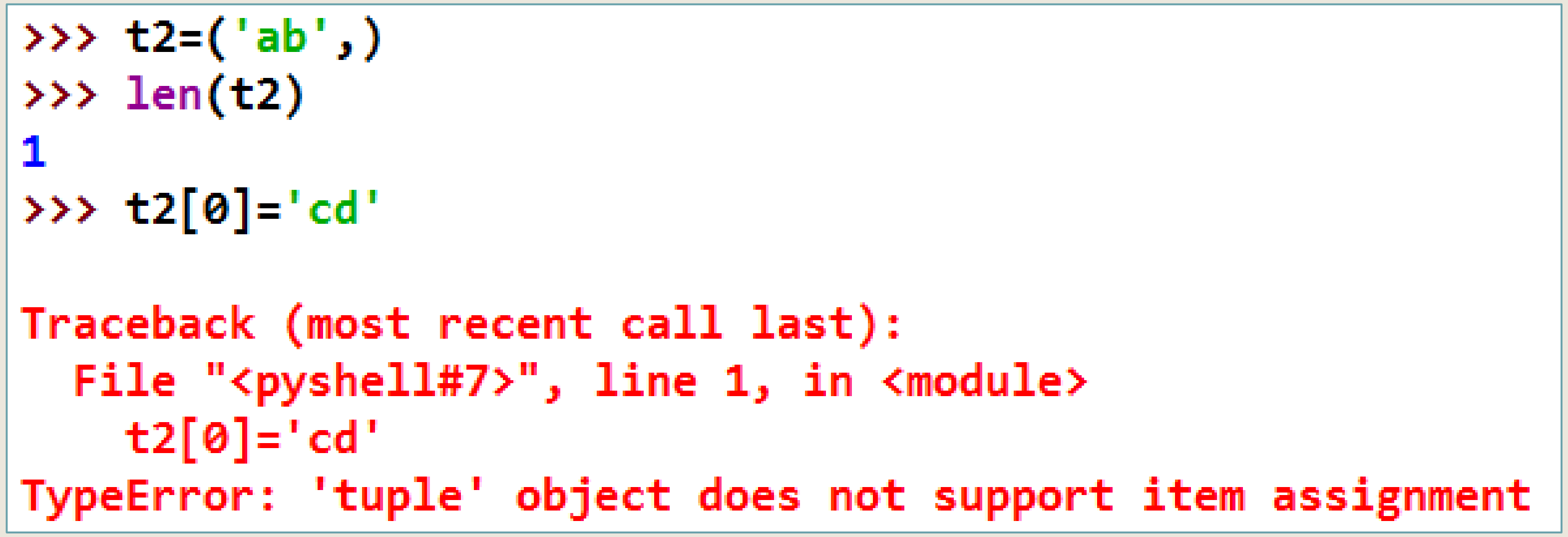
This is a tuple



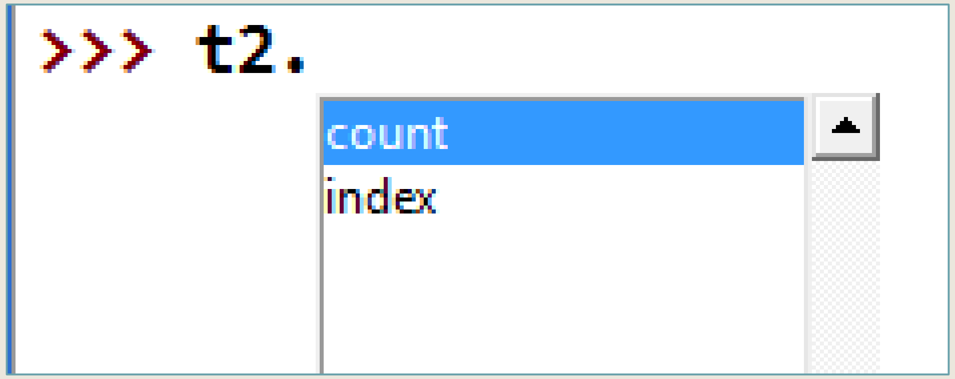
```
>>> type(t)
<type 'int'>
>>> type(t2)
<type 'tuple'>
```

Python Tuple

- Immutable type: cannot be changed



```
>>> t2=('ab',)
>>> len(t2)
1
>>> t2[0]='cd'

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    t2[0]='cd'
TypeError: 'tuple' object does not support item assignment
```



```
>>> t2.
```

Only two methods for tuples!

Tuples can be used for multiple assignments

```
>>>(a, b) = (10, 20)
>>>print (a)
10
>>>print (b)
20
```

Assign multiple variables at once

Working with tuples

```
>>> x=(1,2,3)
>>> type(x)
<class 'tuple'>
```

In many ways tuples behave like lists or dictionaries

```
>>> x+(4,5,6)
(1, 2, 3, 4, 5, 6)
```

concatenation

```
>>> x*2
(1, 2, 3, 1, 2, 3)
```

repetition

```
>>> x[2]
3
```

slicing

```
>>> len(x)
3
```

length

```
>>> 3 in x
True
```

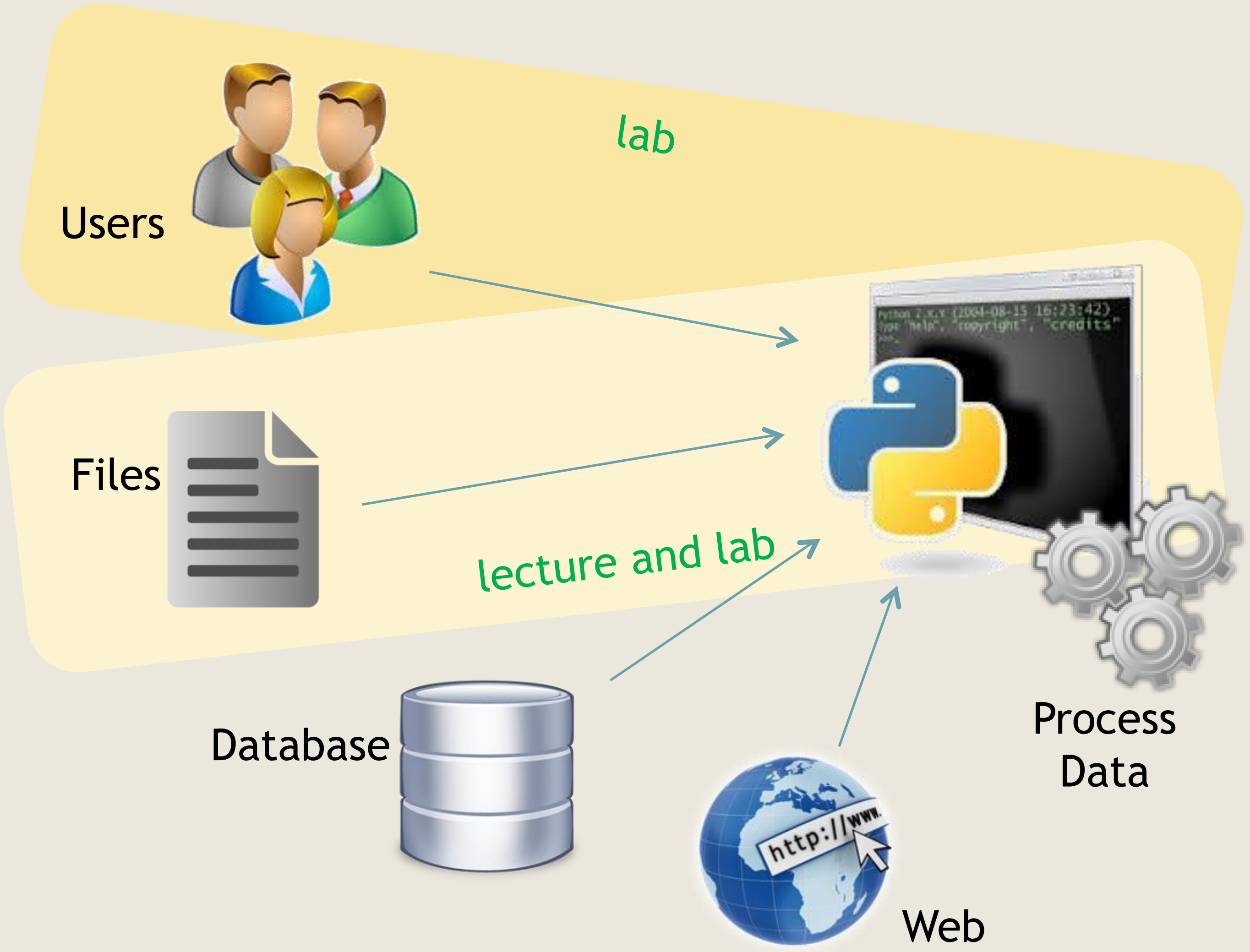
membership

Usage of Tuples

- Good for multiple assignments
  - *unpacking object data into many variables at once*
- Good for grouping things together and passing around as single object
- “Lists” that won't change
  - *fixed number of items*
  - *if we see a tuple we know collection won't change!*
- Provides integrity – know how many objects are needed
- Argument passing, fixed size data records

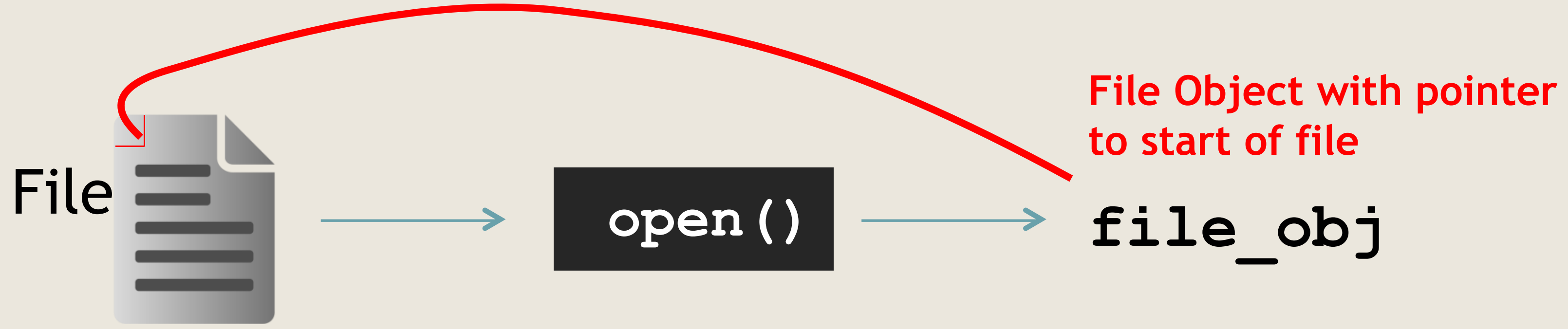
External data 1: Reading Files

Python External Data



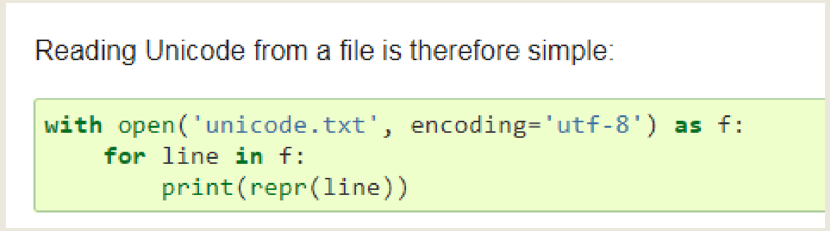
Creating a File Object: open()

- **file_obj = open('filename', 'mode')** # modes 'r' read, 'w' write, 'a' append



- File object Methods
  - file_obj.<tab>
- Close File
  - **file_obj.close()**

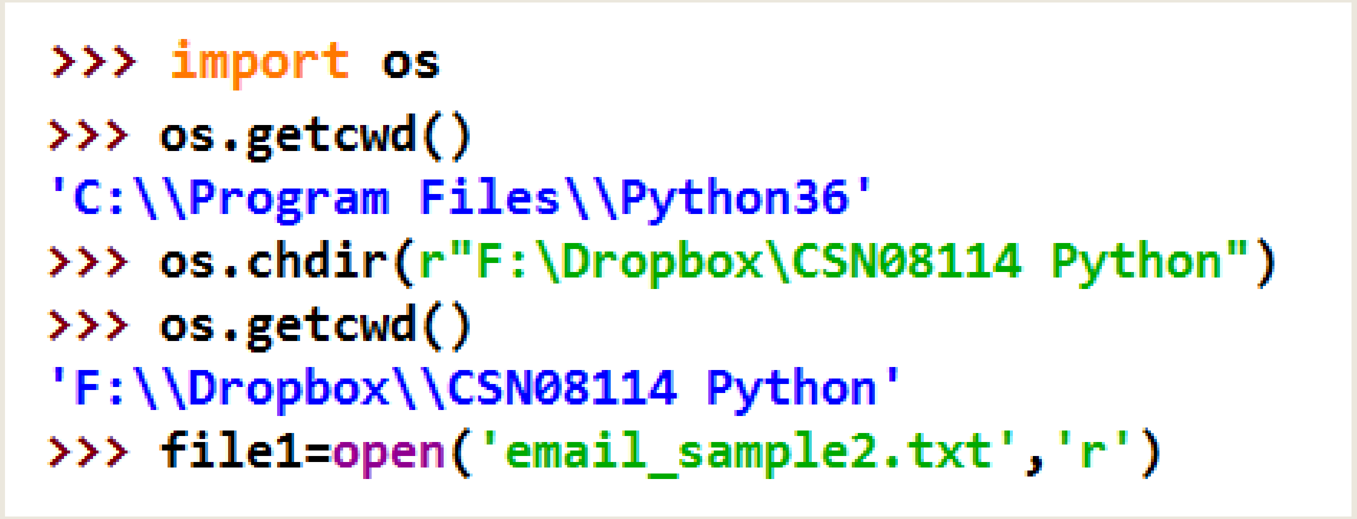
close removes link to file reclaims memory and flushes buffer


Reading Unicode from a file is therefore simple:

```
with open('unicode.txt', encoding='utf-8') as f:
    for line in f:
        print(repr(line))
```

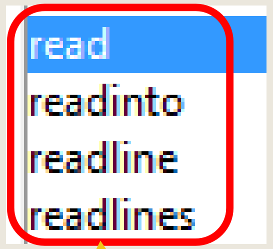
A note on paths

- IDLE always has a current working directory
- Usually C:\..\Python37
- If you want to open a file that's stored somewhere else, either specify the path as part of the filename or change the working directory with os.chdir().

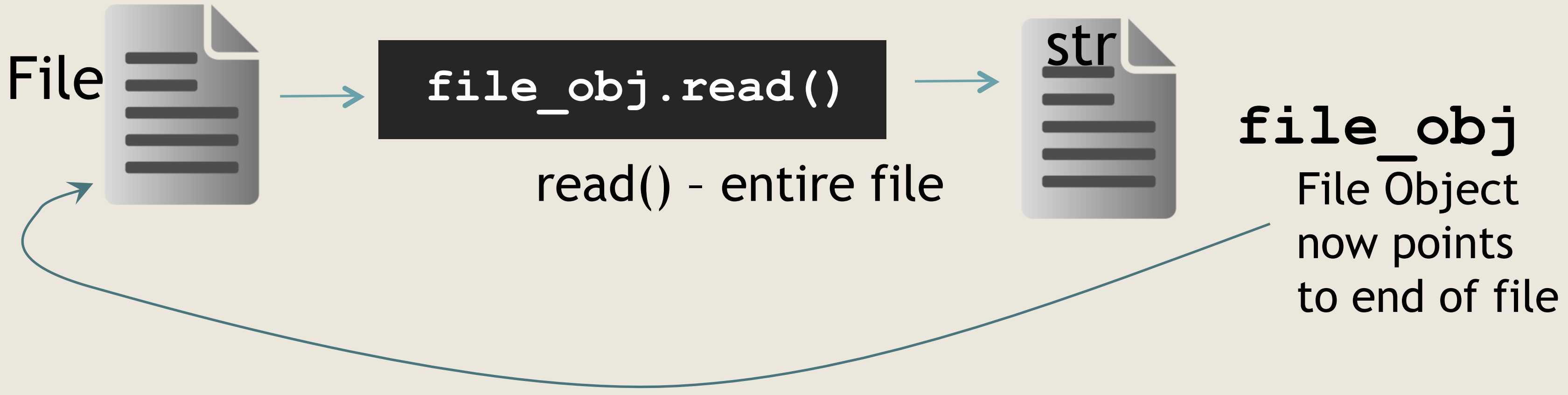


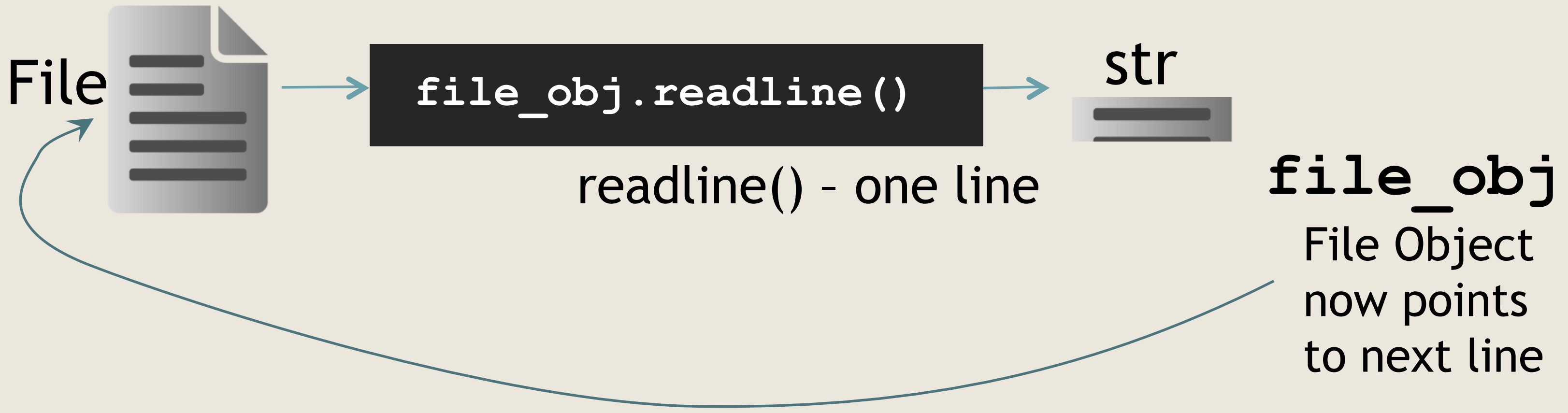
```
>>> import os
>>> os.getcwd()
'C:\\Program Files\\Python36'
>>> os.chdir(r"F:\Dropbox\CSN08114 Python")
>>> os.getcwd()
'F:\\Dropbox\\CSN08114 Python'
>>> file1=open('email_sample2.txt','r')
```

Reading from File

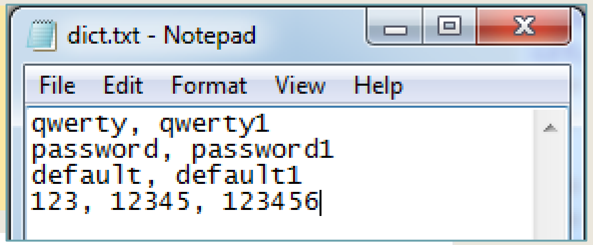


Several ways to
read data from file



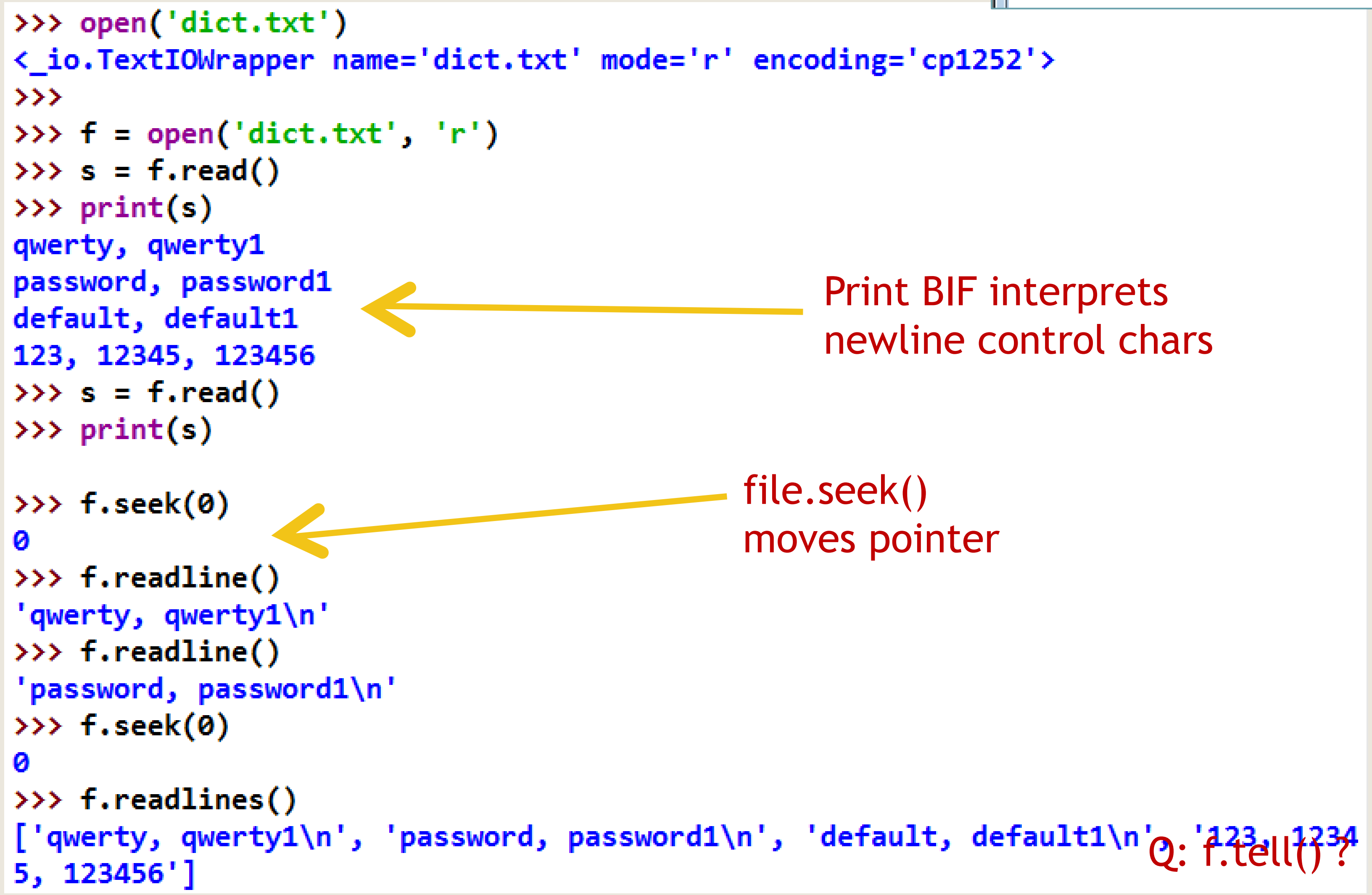


Reading from File



```
qwerty, qwerty1
password, password1
default, default1
123, 12345, 123456
```

- file.read() reads file content into a string
- file.readline() reads one line
- file.readlines() reads file content into a list



```
>>> open('dict.txt')
<_io.TextIOWrapper name='dict.txt' mode='r' encoding='cp1252'>
>>>
>>> f = open('dict.txt', 'r')
>>> s = f.read()
>>> print(s)
qwerty, qwerty1
password, password1
default, default1
123, 12345, 123456
>>> s = f.read()
>>> print(s)

>>> f.seek(0)
0
>>> f.readline()
'qwerty, qwerty1\n'
>>> f.readline()
'password, password1\n'
>>> f.seek(0)
0
>>> f.readlines()
['qwerty, qwerty1\n', 'password, password1\n', 'default, default1\n', '123, 1234
5, 123456']
```

Print BIF interprets newline control chars

file.seek() moves pointer

Q: f.tell() ?

Python File Object Iteration

- For reading lines from a file, you can also loop over the file object with **`for…in`**

```
for line in file_obj:
    do something with the line
```

```
>>> f = open('dict.txt', 'r')
>>> for line in f:
        print (line, end='')
qwerty, qwerty1
password, password1
default, default1
123, 12345, 123456
```

Why use this rather than read entire file at once?

Security: Hashes

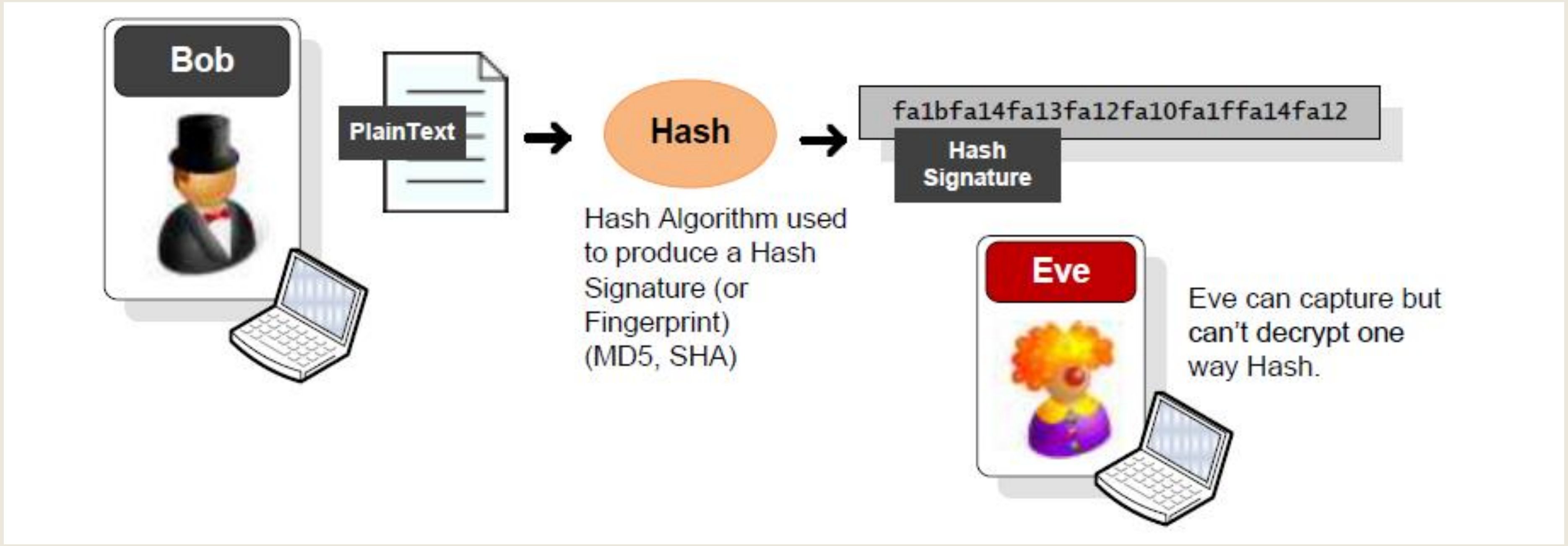
…

What is a hash?

Hash algorithm – one way encryption

- A Hash algorithm generates a signature or Hash fingerprint for a given input.

MD5 and SHA are common hash algorithms
SHA is a whole family: SHA-1, SHA-256, SHA512 etc


Bob
PlainText
Hash
fa1bfa14fa13fa12fa10fa1ffa14fa12
Hash Signature
Hash Algorithm used to produce a Hash Signature (or Fingerprint) (MD5, SHA)
Eve
Eve can capture but can't decrypt one way Hash.

Interactive hash generator: https://asecuritysite.com/encryption/md5

Uses of Hashes

- Authentication
- Integrity of data and messages
  - *Compare original hash signature of a file/message with later hash: know if the file/message has been changed.*
- Password storage and transmission
- Hashes are not reversible
  - *But can potentially be cracked*

Properties of Hashes

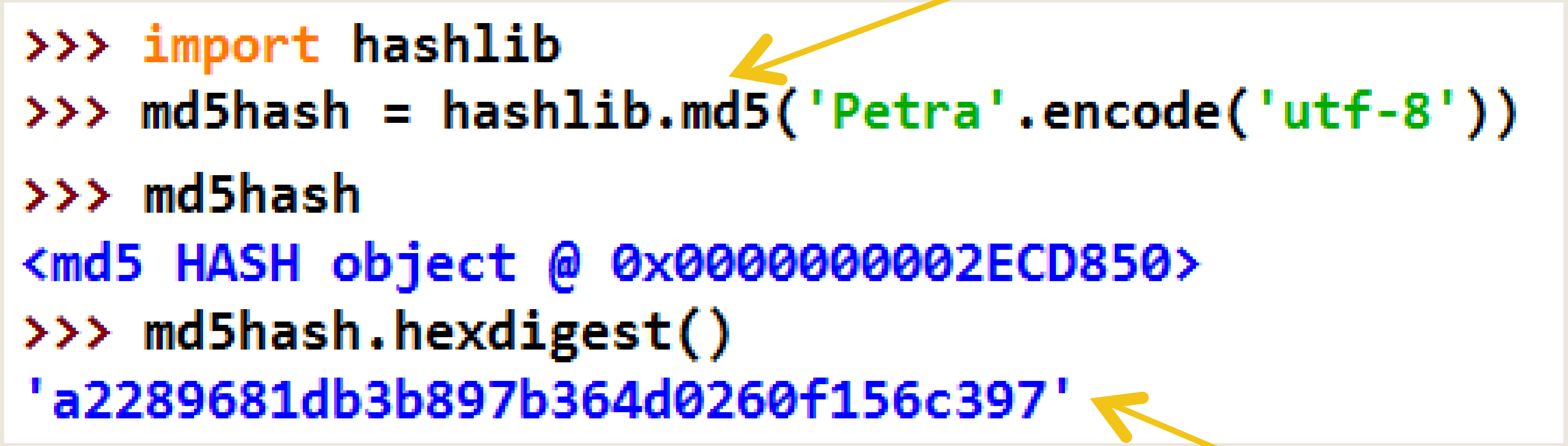
- Hashes are not reversible
  - *But can potentially be cracked*

- All hashes generated by same algorithm are the same length
  - *Shorter than a file or message (128, 256, 512 bits are common)*
- Hashes are usually written in hex
- Hashes are generally unique (similar to fingerprints)
- Small change in message -> big change in hash value

Hashes in Python

- **hashlib** library contains common hash functions

- Example: **MD5 Hash**



```
>>> import hashlib
>>> md5hash = hashlib.md5('Petra'.encode('utf-8'))
>>> md5hash
<md5 HASH object @ 0x000000002ECD850>
>>> md5hash.hexdigest()
'a2289681db3b897b364d0260f156c397'
```

md5 Function in **hashlib** library creates MD5 Hash object from encoded plaintext input

.hexdigest() method gives Hash Signature in Printable (hex) format

Scripting for Cybersec & Networks

Security: Password Hashes

Password Hashes: use for offline password recovery

Windows operating systems store passwords as hashes in the Security registry hive.

Windows Login user authentication:
hash is created for password entered and compared against hash for username in registry.

Passwords cannot be retrieved directly from hashes.
Hashes are one-way encryption and cannot be reversed!

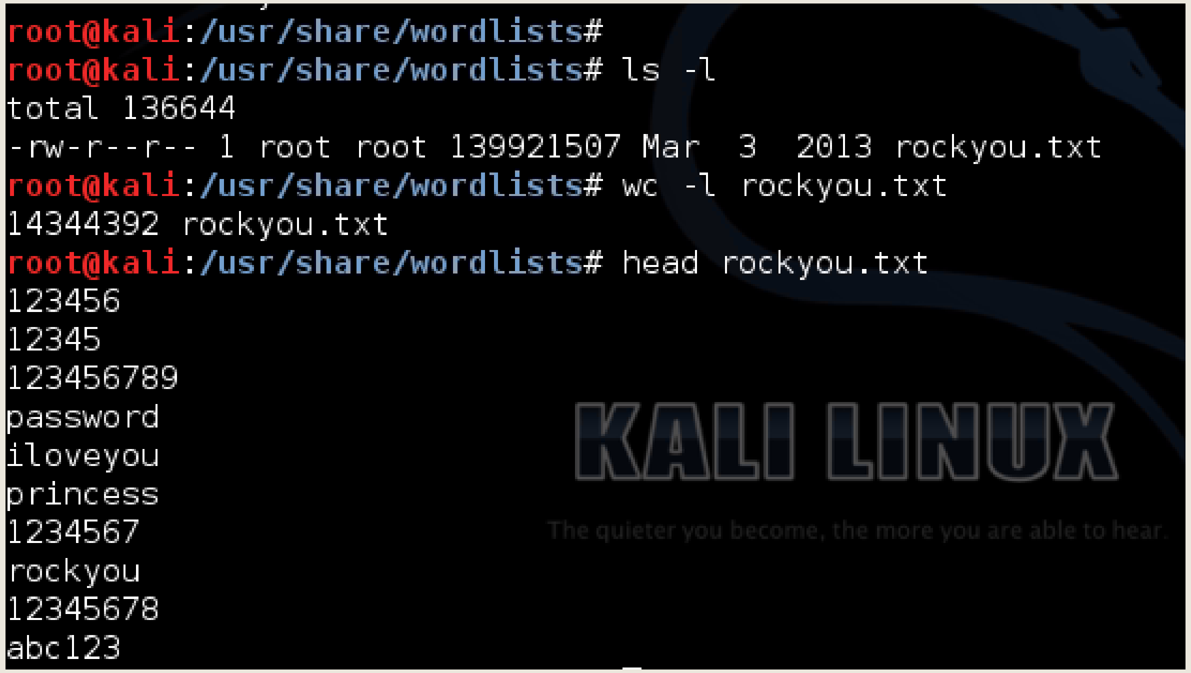
To recover the passwords from the stored hashes, compute hashes from possible passwords lists and compare

Password Recovery - Offline Techniques

Brute Force the Keyspace	Dictionary Attack	Hybrid Attack	Hash Tables
• Try every possible combination of characters	• Use wordlists and combinations of words and symbols	• Algorithms to use words/ symbols and brute force likely parts of keyspace	• **Rainbow Tables** - Precomputed hash lookup tables stored in efficient lookup structure

Password Recovery - Offline Techniques

- Use wordlists of common passwords for Dictionary Attack
  - *Also used by Cracking Tools*
- Well-known list is rockyou.txt
  - *download e.g. from github*
  - *Bundled e.g. with kali*



```
root@kali:/usr/share/wordlists#
root@kali:/usr/share/wordlists# ls -l
total 136644
-rw-r--r-- 1 root root 139921507 Mar  3  2013 rockyou.txt
root@kali:/usr/share/wordlists# wc -l rockyou.txt
14344392 rockyou.txt
root@kali:/usr/share/wordlists# head rockyou.txt
123456
12345
123456789
password
iloveyou
princess
1234567
rockyou
12345678
abc123
```

Go to www.menti.com and use code 15 22 57

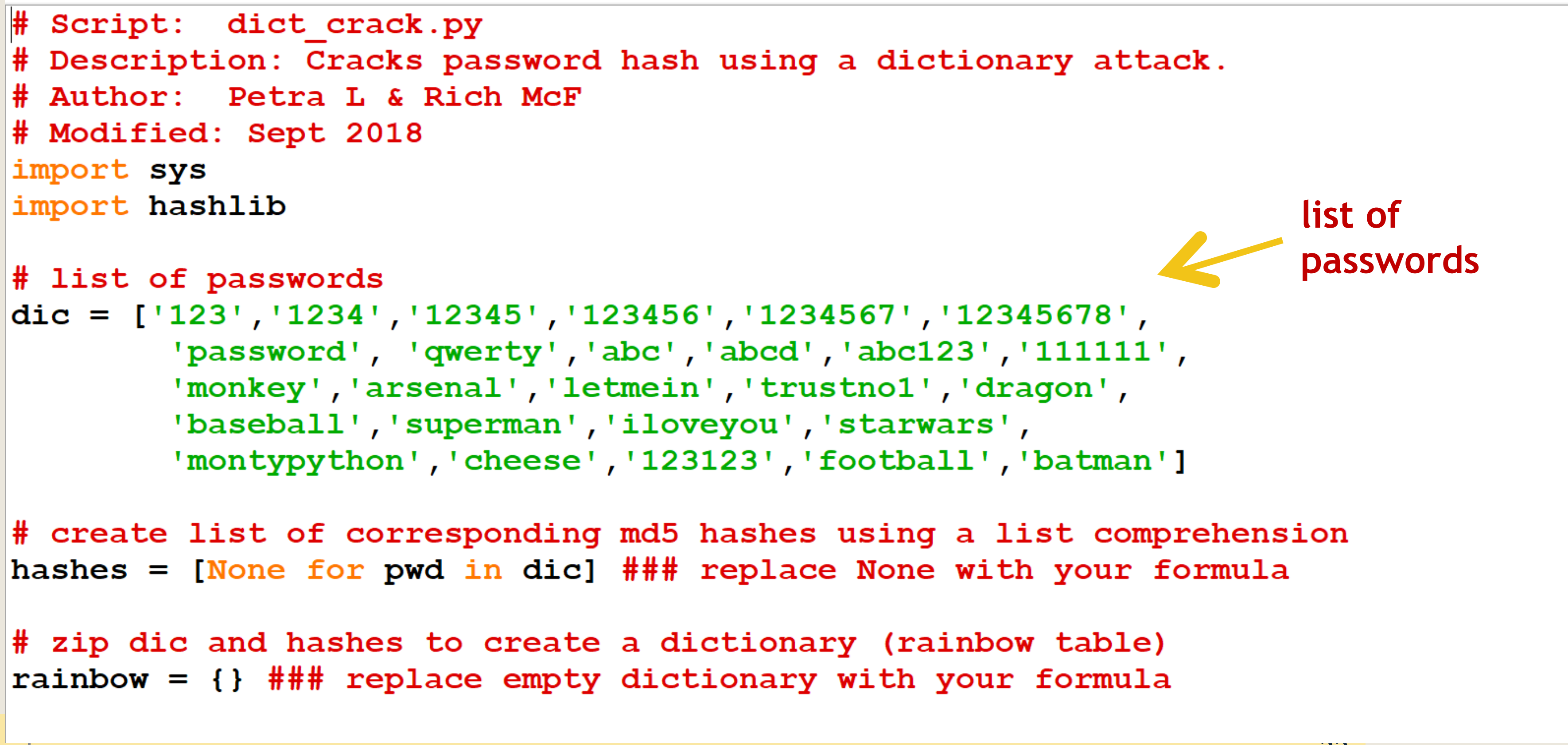
Practical Lab 03

- Password recovery (cracking) script

Password Recovery Script (Lab 3)

- **Dictionary Attack** on password hash
- `dict_crack.py`

Hashing and dictionary creation



```
# Script:   dict_crack.py
# Description: Cracks password hash using a dictionary attack.
# Author:   Petra L & Rich McF
# Modified: Sept 2018
import sys
import hashlib

# list of passwords
dic = ['123','1234','12345','123456','1234567','12345678',
        'password', 'qwerty','abc','abcd','abc123','111111',
        'monkey','arsenal','letmein','trustno1','dragon',
        'baseball','superman','iloveyou','starwars',
        'montypython','cheese','123123','football','batman']

# create list of corresponding md5 hashes using a list comprehension
hashes = [None for pwd in dic] ### replace None with your formula

# zip dic and hashes to create a dictionary (rainbow table)
rainbow = {} ### replace empty dictionary with your formula
```

list of passwords

Password Recovery Script (Lab 3)

Hash Signature
Recovery function called with argument of password hash

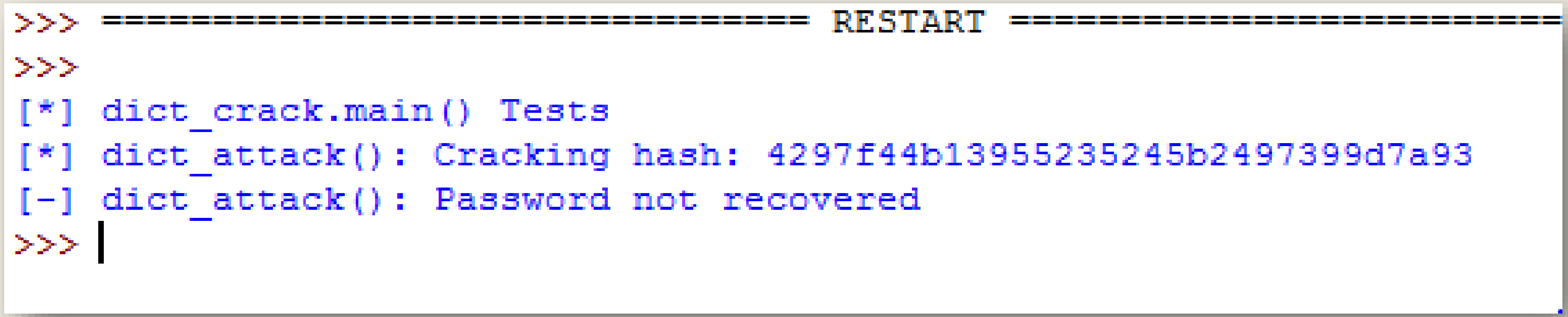
Look up hash in dictionary

Hash
Signatures
Test case(s)

```python
def dict_attack(passwd_hash):
    """Checks password hash against a dictionary of common passwords"""

    print (f'[*] Cracking hash: {passwd_hash}')


    passwd_found = None ### replace None with a look up using .get() on rainbow

    if passwd_found:
        print (f'[+] Password recovered: {passwd_found}')
    else:
        print (f'[-] Password not recovered')

def main():
    print('[dict_crack] Tests')
    passwd_hash = '4297f44b13955235245b2497399d7a93'
    dict_attack(passwd_hash)

if __name__ == '__main__':
        main()
```

Password Recovery Script (Lab 3)

- Run the code:



```
>>> ================================ RESTART =========================
>>>
[*] dict_crack.main() Tests
[*] dict_attack(): Cracking hash: 4297f44b13955235245b2497399d7a93
[-] dict_attack(): Password not recovered
>>> |
```

…not surprising as code doesn't do anything yet…

Password Recovery Script (Lab 3)

- How can we test?
  - Test in Interpreter… Add a known test case and use the known password?
  - Test loop with known test case and the known password first